Asynchronous Session Types
– Exceptions and Multiparty Interactions –

Marco Carbone[1], Nobuko Yoshida[2], and Kohei Honda[3]

[1] IT University of Copenhagen
[2] Imperial College London
[3] Queen Mary, University of London

Abstract. Session types are a formalism for structuring communication based on the notion of *session*: the structure of a conversation is abstracted as a type which is then used as a basis of validating programs through an associated type discipline. While standard session types have proven to be able to capture many real scenarios, there are cases where they are not powerful enough for describing and validating interactions involving more complex scenarios. In this note, we shall explore two extensions of session types to *interactional exceptions* and *multiparty session* in presence of asynchronous communication.

1 Introduction

Recent years have seen the emergence of a new style of distributed software system, called *web services*, designed to support interoperable machine-to-machine interaction over a network, using the infrastructure of the world-wide web. These interactions can make up a sophisticated application whose major mode of computation is communication among distributed computing entities. The advent of web services, together with other trends such as the emergence of multi-core processors and ubiquitous computing, is contributing to a shift in the software development paradigm, where communication and concurrency are a norm rather than exceptions. This new paradigm however still lacks a mature programming methodology. Programming communication and concurrency is harder than sequential programming, as it exposes programmers and designers to the new level of complexity including composition of communication behaviours, deadlock, livelock, and diverse forms of partial failure. Thus, web services pose major technical challenges in programming methodologies. At one of the most basic level, these challenges may be summarised as follows: i) we should be able to describe communication-centred behaviour clearly, accurately and in a modular way; ii) we should be able to validate and detect critical properties of programs with respect to their communication behaviour; and iii) we should be able to control run-time behaviour of programs including their composition.

Session types [16, 8] are types for structuring communication and have been studied over the last decade for a wide range of process calculi and programming languages. In session types, the notion of *session* becomes central: communication-centred applications exhibit a highly structured sequence of interactions involving, for example,

branching and recursion, which as a whole form a natural unit of conversation, or session. The structure of a conversation is abstracted as a type which is then used as a basis of validating programs through an associated type discipline.

While original session types have proven to be very simple and concise and able to capture many real scenarios, there are cases where they are not powerful enough for describing and validating interactions involving more complex conversation patterns or, they are just too complex to use, making the design stage harder than it should be.

In this lecture note, we shall explore two extensions of the foregoing theories on session types to (1) interactional exceptions [5] and (2) multiparty sessions [10, 1, 2] in an asynchronous setting which often arise in practical communication-centred applications.

1. Interactional exceptions, an interactional asynchronous generalisation of structured exceptions, allow communicating peers to asynchronously and collaboratively escape from the middle of a dialogue (session) and reach another in a coordinated fashion. New exception types guarantee communication safety and offer a precise type-abstraction of advanced conversation patterns found in practice.
2. Multiparty sessions extend binary sessions to multiparty, asynchronous sessions. In multiparty sessions, interactions involve multiple peers which are directly abstracted, at type level, as a global scenario. Global types retain the friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. The fundamental properties of the session type discipline such as communication safety, progress and session fidelity hold for general n-party asynchronous interactions.

The remainder of this paper is structured as follows: Section 2 gives the common notation for the following sections; Section 3 introduces an extension of standard session types to an exception mechanism similar to the one of imperative programming languages; Section 4 addresses an extension of binary session types to multiparty; and Section 5 contains some concluding remarks.

2 Notation

Works on session types tend to adopt different notation depending on the specific problem modelled. Due to the nature of this lecture note which addresses two extensions of session types to asynchronous interactional exceptions and multiparty sessions, we shall set some common notation, trying not to deviate from the original one used in [8].

Shared channels represent those names/channel that are public i.e. known by any process in the modelled system. Shared channels are also known as *public* or *service* channels and are denoted by $a, b, c, \ldots$.

A session can be seen as a conversation between some parties which share common session identifiers (or channel names) used for communicating values. *Session channels* are denoted by k (as in the original work on session types [8]) or $s, t, r \ldots$.

Session channels can also be polarised (s^+, s^-) i.e. a polarity is assigned to the channel in order to identify the side of the (binary) session [6]. We shall use polarised channels when implementing asynchronous exceptions for telling from/to who an exception has been raised.

Term	Symbol	Note
Public channels (or *shared* or *service* channels)	$a, b, \ldots$	
Session channels	$k, s, t, r \ldots$	
Polarised Session channels	κ, λ	$\kappa \in \{s^+, s^-\}$
Variables	x, y, z	
Process Term Variables	X, Y	
Public channel or Variable	$u, u', \ldots$	

Table 1. Notation on Session Types

Formally, s^p is a *polarised session channel* with p ranging over polarities $\{+, -\}$. We define the dual of a polarised channel s^p as $\overline{s^+} = s^-$ and $\overline{s^-} = s^+$. Polarised channels are denoted by Greek letters κ and λ.

The letters x, y, z denote *variables* while $X, X', \ldots$ are *term variables*.

Table 1 resumes the notation used in this lecture note.

3 Interactional Exceptions in Session Types

3.1 Preview on Interactional Exceptions

According to a Wikipedia entry, an *exception (handler)* is

> *"... a programming language construct [...] designed to handle the occurrence of a condition that changes the normal flow of execution" [18].*

Structured exceptions in modern programming languages such as Java and C♯ allow a thread of control in a block (often designated as "`try` block") to get transferred to another block (exception handler, "`catch` block"), when a system or user raises an exception. Their central merit is to enable a dynamic escape from a block of code to another (like `goto`), but in a controlled and structured way (unlike `goto`). They are useful not only for error-handling but, as suggested by the citation above, also for a flexible control flow while preserving well-structured description and type-safety.

In this section, we address the notion of structured exceptions for distributed, concurrent, asynchronously communicating programs based on session types motivated by collaboration with industry partners in web services [17] and financial protocols [12]. These two application domains contain a wealth of structured conversation patterns arising from practical needs [9], and many of these patterns crucially rely on dynamic escape: a conversation is interrupted by a special communication action, after which all peers move to a different stage. Hence, an exception affects not only a sequential thread but also a collection of parallel processes; and an escape needs to move into another dialogue in a concerted manner. The distinguishing feature of these exceptions in comparison with their traditional counterpart is that they demand not only local but also coordinated actions among communicating peers. We call such exceptions, *interactional exceptions*.

Example 1 (Asynchronous and Nested Escapes) We conclude this preview, with an example scenario based on financial protocols. Suppose a seller Seller wishes to sell a product to a buyer Buyer such that:

> 1. Seller repeats sending quotes without waiting for an acknowledgement;
> 2. if Buyer accepts one of the quotes, the loop terminates and the conversation moves to another stage for completing the transaction.

The conversation pattern above contains an asynchronous escape from one part of a conversation to another: after Buyer aborts, both participants should move together to another part of the conversation. The protocol can become more complex, involving other parties e.g. Buyer and Seller negotiate the price through a broker Broker:

> 1. Buyer initiates a conversation with Broker;
> 2. as a result, Broker initiates a conversation with Seller, and starts brokering between Buyer and Seller, to reach a successful transaction;
> 3. if an exceptional situation arises between steps 1 and 2 (e.g. a legal issue), Buyer or Broker aborts and they together move to a quitting dialogue;
> 4. on the other hand, if there is an exceptional circumstance during 2, then there is an exception dialogue involving all of Broker, Seller and Buyer.

Above, an exception handling at Broker is *nested*, whose later, or inner, exception handling (4, involving all three parties) supersedes the earlier, or outer, one (3, involving only Broker and Seller). As a conversation evolves, more communication peers may be involved, making it necessary to coordinate more parties when an exception is raised.

3.2 The π-Calculus with Asynchronous Sessions and Interactional Exceptions

Syntax. The syntax of (static) processes (denoted by $P, Q, R, \ldots$) written by programmers is given by the following grammar:

$$
\begin{array}{lrlr}
P ::= & * c(\lambda)[P, Q] & \text{(accept)} & \mid \overline{c}(\lambda)[\widetilde{\kappa}, P, Q] \quad \text{(request)} \\
& \mid \kappa?(x).\ P & \text{(input)} & \mid \kappa!\langle e \rangle.\ P \quad \text{(output)} \\
& \mid \kappa \rhd \{l_i\colon P_i\}_{i \in I} & \text{(branch)} & \mid \kappa \lhd l.\ P \quad \text{(select)} \\
& \mid P \mid Q & \text{(par)} & \mid \textbf{if } e \textbf{ then } P \textbf{ else } P \quad \text{(cond)} \\
& \mid \mathbf{0} & \text{(inact)} & \mid (\nu a)\ P \quad \text{(resServ)} \\
& \mid X & \text{(termVar)} & \mid \mu X.\ P \quad \text{(recursion)} \\
& \mid \textbf{throw} & \text{(throw)} &
\end{array}
$$

$$
e ::= a \mid \mathsf{tt} \mid \mathsf{ff} \mid e \ \mathsf{and}\ e \mid \neg e \mid \ldots \qquad c ::= a \mid x \qquad \kappa, \lambda ::= s^p
$$

Above, the accept term $*a(\lambda)[P, Q]$ is a replicated process with shared channel a, polarised session channel κ, *default process* P and *exception handler* Q. The term denotes a service a which, when invoked, establishes a fresh session channel κ and behaves as process P, possibly followed by Q if an exception takes place. Service a is replicated (available in many copies) according to the *Service Channel Principle* (SCP) [4]:

Definition 1 (Service Channel Principle (SCP)). *Invocation channels are always available. Therefore, they can be shared and invoked repeatedly.*

Dually, a request $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$ interacts with a service via c and establishes a fresh session λ, with its *default process* P and *handler* Q. Because shared channels can be passed, we allow for c to be a variable e.g. it could be bound by a prefixing input.

The session channels $\tilde{\kappa}$, containing already established sessions which the handler Q gets associated with, have a pivotal rôle: in the case P raises an exception, any other handler belonging to an embedding accept $*c'(\lambda')[P', Q']$ or request $\overline{c'}(\lambda')[\tilde{\kappa}', P', Q']$ ($\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$ is in P'), such that $\tilde{t}' \subseteq \tilde{t}$, must be discarded. We call vector $\tilde{t}$ a *refinement* in the sense that channels $\tilde{t}'$ in $\tilde{t}$ are refined i.e. a new handler Q replaces the old Q'. We require λ itself to be included in $\tilde{\kappa}$ which is convenient for typing. As an example, in the process:

$$\overline{a}(\kappa)[\kappa, \quad \overline{b}(\lambda)[(\kappa, \lambda), P, Q] \quad , Q']$$

the term $\overline{b}(\lambda)[(\kappa, \lambda), P, Q]$ is a refinement of κ in the sense that once session b is initiated Q becomes the handler for κ and Q' can be discarded.

Process **throw** denotes the throwing of an exception and it usually occurs inside try-catch blokcs. All other constructs are from [8, 4].

Free/bound (term) variables/channels and α-equivalence are standard. $\mathsf{fsc}(P)$, $\mathsf{fn}(P)$ and $\mathsf{fv}(P)$ respectively denote the sets of free session channels, shared channels, and variables in P. We call *program* a process which does not contain free variables or free session channels. We often omit the tailing $\mathbf{0}$.

Syntactic Assumptions. In order to have consistent operational semantics, we stipulate the following syntactic constraints:

1. *(Consistent Refinement)* given $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$, for each $\overline{c'}(\lambda')[\tilde{\kappa}', P', Q']$ occurring in P and any $\kappa_i \in \tilde{\kappa}$, we have $\kappa_i \in \tilde{\kappa}'$ implies $\tilde{\kappa} \subseteq \tilde{\kappa}'$ (for consistent refinement). Further, such a refinement never occurs inside a handler (otherwise we have ambiguity when launching a handler);

2. recursions is *guarded*, i.e. P in $\mu X.\ P$ is prefixed by an input, output, branch, select or conditional; moreover, a free term variable never occurs free in $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$;

3. the term (accept) never occurs under an input/output/recursion prefix nor inside a default process or handler thus protecting its availability from exceptions;

4. **throw** never occurs inside a handler hence preventing a handler from throwing a further exception in the same session.

The above restrictions could be enforced with the typing system but have been separated for the sake of presentation.

Example 2 (Asynchronous Escape) We can write the first part of the example in Section 3.1 as:

$$\begin{array}{ll}
\texttt{Buyer} = \overline{\texttt{chSeller}}(s^+)[\ s^+, & \texttt{Seller} = *\texttt{chSeller}(s^-)[\\
\quad \mu X.\ s^+?(y).\ \textbf{if}\ \texttt{ok}(y)\ \textbf{throw else}\ X, & \quad \mu X.\ s^-!\langle \texttt{quote} \rangle.\ X\ , \\
\quad s^+!\langle \texttt{card} \rangle.\ s^+?(z)\ ] & \quad s^-?(y_2).\ s^-!\langle \texttt{time} \rangle\ ]
\end{array}$$

Buyer keeps on reading messages on s^+ until condition $\texttt{ok}(y)$ is met and then it throws an exception. Seller, instead, is in an infinite loop where it persistently sends a quote over channel s^- (we assume `quote` changes over time). When the exception is raised the handlers are run: Buyer will send the credit card details `card` and Seller will acknowledge on channel s^- with the current time.

Example 3 (Nested Escapes) The second part of Example 1, can be represented in the calculus as (Seller remains unchanged):

$$\begin{array}{ll}
\texttt{Buyer} = \overline{\texttt{chBroker}}(t^+)[\ t^+, & \texttt{Broker} = *\texttt{chBroker}(t^-)[\ t^-, \\
\quad t^+!\langle \texttt{id} \rangle. & \quad\quad t^-?(x).\ \textbf{if}\ \texttt{bad}(x)\ \textbf{then throw else} \\
& \quad\quad \overline{\texttt{chSeller}}(s^+)[\ (s^+, t^-), \\
\quad \mu X.\ t^+?(y).\ \textbf{if}\ \texttt{ok}(y)\ \textbf{throw else}\ X, & \quad\quad\quad \mu X.\ s^+?(x).\ t^-!\langle x + 10\% \rangle.\ X, \\
& \quad\quad\quad t^- \triangleleft l_1.\ t^-?(y_2).\ s^+!\langle y_2 \rangle. \\
\quad t^+ \triangleright \{\ l_1 : t^+!\langle \texttt{card} \rangle.\ t^+?(z), & \quad\quad\quad s^+?(y_3).\ t^-!\langle y_3 \rangle\quad], \\
\quad\quad l_2 : P_{\texttt{abort}}\}\quad] & \quad\quad t^- \triangleleft l_2.\ R_{\texttt{abort}}\quad]
\end{array}$$

Buyer first sends its identity `id` and then Broker throws an exception or proceeds by invoking Seller based on `bad(id)`. In the first case, process $t^- \triangleleft l_2.\ R_{\texttt{abort}}$ in the outermost handler selects the l_2 branch on Buyer's handler and proceeds with abortion (conversation between $P_{\texttt{abort}}$ and $R_{\texttt{abort}}$). In the other case, Seller is invoked and the protocol proceeds as in Example 2 with Broker forwarding messages and increasing quotes by 10%. When Buyer decides to accept a quote, the innermost handler is run by Broker which selects the l_1 conversation in Buyer's handler and forwards the exception to Seller. Then Broker forwards messages, successfully completing the transaction.

Semantics. We shall now define the semantics of asynchronous sessions [3, 7, 10, 5] with exception handling and exception propagation. Further we ensure that processes always carry out their conversation at properly matching levels (for example when a default process sends a message, a receiving peer may throw an exception before the message arrives, making it no longer relevant), by annotating message queues, hence in effect messages in them, with exception levels.

In order to implement asynchrony of communication (both for messages and exception propagation), we need to extend the grammar of programs with extra syntactic terms called *runtime processes* [3, 7, 10, 5]:

$$\begin{array}{llll}
P ::= \ \ldots \mid (\nu s)\ P & \text{(resSess)} & \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L & \text{(queue)} \\
\quad \mid \textbf{try}\{\ P\ \}\ \textbf{catch}\ \{\ \tilde{\kappa} : Q\ \} & \text{(try-catch)} & \mid \tilde{\kappa}[\![P]\!] & \text{(wrap)}
\end{array}$$

$$L ::= \quad \epsilon \mid h :: L \qquad\qquad h ::= \quad l \mid a \mid \ \mathsf{tt}\ \mid\ \mathsf{ff}\ \mid \dagger$$

The *try-catch block* $\textbf{try}\{\ P\ \}\ \textbf{catch}\ \{\ \tilde{\kappa} : Q\ \}$ is the runtime presentation of a default process and a handler: the default process P in the *try-block* is running during which an

$$\text{(MTry)}\quad P \searrow (P', S) \quad \Rightarrow \quad \mathbf{try}\{\,P\,\}\,\mathbf{catch}\,\{\,\tilde{\kappa} : Q\,\} \searrow \begin{cases} (P', S) & \text{if } \tilde{\kappa} \subseteq S \\ (\tilde{\kappa}[\![Q]\!] \mid P', S \cup \tilde{\kappa}) & \text{otherwise} \end{cases}$$

$$\text{(MWrap)}\quad \tilde{\kappa}[\![Q]\!] \searrow (\tilde{\kappa}[\![Q]\!], \emptyset)$$

$$\text{(MPar)}\quad P \searrow (P', S_1) \text{ and } Q \searrow (Q', S_2) \quad \Rightarrow \quad P \mid Q \searrow (P' \mid Q', S_1 \cup S_2)$$

$$\text{(MNil)}\quad R \searrow (\mathbf{0}, \emptyset) \quad \text{if } R \in \left\{ \begin{array}{l} \text{(inact),(request), (input), (output), (branch),} \\ \text{(select), (cond), (recursion), (throw)} \end{array} \right\}$$

Table 2. Rules for Meta Reduction

exception on channels $\tilde{\kappa}$ can be thrown, which terminates P and launches the handler Q in the *catch-block*. When this Q is launched, it becomes a *wrapped process* (or, simply, a *wrap*) $\tilde{\kappa}[\![Q]\!]$, making Q immune to an exception notification at the same or upper levels (note such notifications can come due to asynchrony).

In order to formalise order-preserving asynchronous message passing, we use a directed message queue $\kappa \hookrightarrow_\phi \overline{\kappa} : L$ [3, 7, 5, 10], where κ (source) and $\overline{\kappa}$ (target) are two dual polarised session channels. ϕ ranges over natural numbers, describing the level of the exception at which messages in the queue are to be delivered (e.g. to a try-block or a wrap). This will also be relative to the current position of the queue, which is allowed to move inside/outside try-catch blocks and wraps. We do not need to consider the level of a sender, since this level is recorded by the number of the exception messages † inside a queue. We often write $\kappa \hookrightarrow \overline{\kappa} : L$ for $\kappa \hookrightarrow_0 \overline{\kappa} : L$. The list $L :: h$ is obtained by extending L with an extra tail element h. Given the list $L = L' :: h'$, we stipulate that inserting a message h in L will result into $h :: L$ while removing an element from L will result into L'.

Session restriction $(\boldsymbol{\nu} s)\ P$ is standard. Free variables and channels are extended to run-time processes.

Meta Reduction. We introduce an extra relation on processes called *meta reduction*, for dearling with sudden termination of try-blocks due to the throwing of an exception. *Meta reduction*

1. erases the remaining activity of the default process in the try-block;
2. propagates exceptions to the try-catch blocks inside the try-block; and
3. leaves wrapped processes as they are.

In traditional structured exceptions as found in Java or C++, an exception completely erases the try-block and lets the handler run in the same state. In our calculus, concurrently running threads inside a try-block may have conversations (sessions) with other agents. Erasing them would make conversations inconsistent, thus an exception is thrown in each of them.

Meta reduction $\searrow$ is the minimum relation satisfying the rules given in Table 2. A reduction $P \searrow (P', S)$ says that the initial process P is transformed into process P', the result of erasing and wrapping; and S denotes session channels via which we

(Init) $*a(s^-)[P, Q] \mid C[\overline{a}(s^+)[\tilde{\kappa}, P', Q']] \longrightarrow$
$$*a(s^-)[P, Q] \mid (\nu s) \begin{pmatrix} \mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, s^- : Q \,\} & \mid & s^- \hookrightarrow_0 s^+ : \epsilon \mid \\ C[\mathbf{try}\{\, P' \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q' \,\}] & \mid & s^+ \hookrightarrow_0 s^- : \epsilon \end{pmatrix}$$

(Out) $\kappa!\langle e \rangle.\ P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : (v :: L) \qquad (e \downarrow v)$

(In) $\kappa?(x).\ P \mid \overline{\kappa} \hookrightarrow_0 \kappa : (L :: v) \longrightarrow P\{v/x\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : L$

(Sel) $\kappa \lhd l.\ P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : (l :: L)$

(Bra) $\kappa \rhd \{l_i : P_i\}_{i \in I} \mid \overline{\kappa} \hookrightarrow_0 \kappa : (L :: l_j) \longrightarrow P_j \mid \overline{\kappa} \hookrightarrow_0 \kappa : L \qquad (j \in I)$

(Con) $P \longrightarrow Q \quad \Rightarrow \quad C[P] \longrightarrow C[Q]$

(If) $\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow P \quad (e \downarrow \mathrm{tt}) \qquad \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow Q \quad (e \downarrow \mathrm{ff})$

(Str) $P \equiv P'$ and $P' \longrightarrow Q'$ and $Q' \equiv Q \quad \Rightarrow \quad P \longrightarrow Q$

(Thr) $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \searrow (R, S) \quad \Rightarrow$
$\mathbf{try}\{\, \mathbf{throw} \mid P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$

(RThr) $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \searrow (R, S) \quad \Rightarrow$
$\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \mid \overline{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa$
$\longrightarrow R \mid \overline{\kappa}_j \hookrightarrow_1 \kappa_j : L \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$

(WVal) $\tilde{\kappa}[\![Q]\!] \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: v) \longrightarrow \tilde{\kappa}[\![Q]\!] \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : L$

(WThr) $\tilde{\kappa}[\![Q]\!] \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \dagger) \longrightarrow \tilde{\kappa}[\![Q]\!] \mid \overline{\kappa}_i \hookrightarrow_1 \kappa_i : L$

(Clean) $P \searrow (R, S),\ (\lambda \in \tilde{\kappa}, \dagger \in L) \Rightarrow$
$\mathbf{try}\{\, P \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : \tilde{Q} \,\} \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa$
$\longrightarrow R \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$

Table 3. Reduction Semantics

should communicate that the exception takes place including the ones of nested try-catch blocks. Rule (MTry) propagates the exception to a nested try-catch block. If the try-block meta reduces to some P' with some set S then $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\}$ will reduce either to (i) P' itself or to (ii) the parallel composition of P' and $\tilde{\kappa}[\![Q]\!]$ with the new set $S \cup \tilde{\kappa}$ ensuring that also channels $\tilde{\kappa}$ will be notified with an exception. Case (i) discards handler Q when another handler for $\tilde{\kappa}$ is already in P while case (ii) happens when there is no refinement of $\tilde{\kappa}$ in P. The mechanism is sound because of the assumption that κ_i are always refined together (cf. syntax). Note that, if the try-block is single-threaded, the meta reduction mechanism is identical to the one of standard exception handling.

Reduction. We now introduce the main reduction rules. Due to the nesting of wraps and try-catch blocks, the reduction is defined using the following reduction contexts:

$$C \ ::= \mathbf{try}\{\, C \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \ \mid\ P \mid C \ \mid\ \tilde{\kappa}[\![C]\!] \ \mid\ (\nu s)\, C \ \mid\ (\nu a)\, C \ \mid\ -$$

Given a context C and a process P, the process $C[P]$ denotes the new process obtained by replacing the whole $-$ in C with P.

The reduction $\longrightarrow$ is the smallest relation generated by the rules in Table 3. (Init)

gives the semantics of session initiation, generating two fresh dual session channels, the associated two empty queues (ϵ denotes the empty string) and the two try-catch blocks $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, s^{-} : Q \,\}$ and $\mathbf{try}\{\, P' \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q' \,\}$. Note that $*a(s^{-})[P, Q]$ is not in a context. This is because we have assumed that *services never appear nested in a try- or a catch-block* as we do not want them to be terminated (following SCP).

(Out) and (Sel) enqueue, respectively, a value and a label at the head of the queue for κ. Symmetrically, (In) and (Bra) dequeue from the tail of the queue. The exception level in the latter two rules is 0, indicating the level of an actual receiver. The exception level of a queue ensures that a message is sent and received at the same level, guaranteeing consistency of communication. This depends on the invariance that the sum of the level of the queue and the number of †'s in the queue before a specific message, determines the depth (the number of wraps) at which the message enqueueing is performed. These rules say that a sending action is never blocked (asynchrony) and that two messages from the same sender to the same channel arrive in the sending order (order preservation).

In (Out,If), $e \downarrow v$ says that expression e evaluates to value v. (Con,Str) are standard.

(Thr) and (RThr) represent the firing of an exception. (Thr) is when **throw** appears top-level in the try-block, i.e. exception is thrown locally; while (RThr) is when a remote exception is received as † in the queue. Eventually, all peers will be notified of the exception by sending † via channels in S generated from P as well as $\tilde{\kappa}$.

(WVal) describes the case when messages at the default level meet a wrapped process and are drained into a sink (i.e. get dequeued but ignored). In (WThr), † meets a wrap and the exception level of the queue is incremented, allowing the queue to enter the wrap. In (Clean), † in the queue reveals the presence of a refinement in P which has now become a wrap due to a local throw. Meta reduction propagates the exception to each parallel process in P and the try-catch block is discarded.

This last step is formally defined by the structural congruence $\equiv$ which plays a key role in treating exceptions and, in particular, moving queues while maintaining their exception levels.

Definition 2 (Structural Congruence). $\equiv$ *is the least congruence relation on processes such that* $(P, |)$ *is a commutative monoid and includes the standard rules for restriction (such as scope extrusion) and recursion and:*

1) $\mathbf{try}\{\, P \mid \lambda \hookrightarrow_{\phi} \overline{\lambda} : L \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \equiv \mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \mid \lambda \hookrightarrow_{\phi} \overline{\lambda} : L \quad (\lambda \in \tilde{\kappa} \Rightarrow \dagger \notin L)$
2) $\tilde{\kappa}[\![P \mid \overline{\lambda} \hookrightarrow_{\phi} \lambda : L]\!] \equiv \tilde{\kappa}[\![P]\!] \mid \overline{\lambda} \hookrightarrow_{\phi} \lambda : L \qquad (\lambda \notin \tilde{\kappa})$
3) $\tilde{\kappa}[\![P]\!] \mid \overline{\kappa}_i \hookrightarrow_{\phi} \kappa_i : L \equiv \tilde{\kappa}[\![P \mid \overline{\kappa}_i \hookrightarrow_{\phi-1} \kappa_i : L]\!]$
4) $\mathbf{try}\{\, (\nu a)\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \equiv (\nu a)\, \mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \qquad (a \notin \mathit{fn}(Q))$
5) $\tilde{\kappa}[\![(\nu a)\, P]\!] \equiv (\nu a)\, \tilde{\kappa}[\![P]\!]$

The first and second rules allow a queue to move into a try-catch block and a wrap respectively. The third rule is applicable when the receiving side of the queue is in $\tilde{\kappa}$: when entering the wrap, ϕ is decreased so that the process inside the wrap can read the value if the level after the decrement is 0. The last two rules open the scope.

Example 4 To illustrate how queue levels work, we consider the following process:

$$\begin{array}{rl} P = & \mathbf{try}\{\ \mathbf{throw} \mid \kappa!\langle 5\rangle\ \}\ \mathbf{catch}\ \{\kappa : \kappa!\langle \mathtt{tt}\rangle\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : \epsilon \mid \\ & \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\ \mathbf{catch}\ \{\overline{\kappa} : \overline{\kappa}?(x)\}\ \ \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \end{array}$$

Process P can reduce to $P' = \kappa[\![\mathbf{0}]\!] \mid \overline{\kappa}[\![\mathbf{0}]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \epsilon \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon$ in different ways.

$$\begin{array}{l} P \longrightarrow\equiv \kappa[\![\kappa!\langle \mathtt{tt}\rangle]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger \mid \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\ \mathbf{catch}\ \{\overline{\kappa} : \overline{\kappa}?(x)\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \\ \quad \longrightarrow\equiv \kappa[\![\mathbf{0}]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : (\mathtt{tt} :: \dagger) \mid \mathbf{try}\{\mathbf{throw} \mid \overline{\kappa}?(x)\ \}\ \mathbf{catch}\ \{\overline{\kappa} : \overline{\kappa}?(x)\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \\ \quad \longrightarrow\equiv \kappa[\![\mathbf{0}]\!] \mid \kappa \hookrightarrow_1 \overline{\kappa} : \mathtt{tt} \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \\ \quad \longrightarrow\equiv \kappa[\![\mathbf{0}]\!] \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \mid \kappa \hookrightarrow_1 \overline{\kappa} : \mathtt{tt} \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \longrightarrow\equiv P' \end{array}$$

In this case, an exception and then `tt` are sent over κ. Finally the exception is delivered to $\overline{\kappa}$ before delivering `tt`. But we can also have:

$$\begin{array}{l} P \longrightarrow\longrightarrow\equiv \mathbf{try}\{\ \mathbf{throw}\ \}\ \mathbf{catch}\ \{\kappa : \kappa!\langle \mathtt{tt}\rangle\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : 5 \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \\ \quad \longrightarrow\equiv \kappa[\![\kappa!\langle \mathtt{tt}\rangle]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : (\dagger :: 5) \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \longrightarrow\longrightarrow\longrightarrow\equiv P' \end{array}$$

Above, 5 is sent over κ and an exception is thrown on $\overline{\kappa}$. In this situation, the system will ignore 5 (discarded by (WVAL)), and deliver `tt` inside the wrap.

Example 5 The following example shows how refinement of an existing exception is handled:

$$R\ =\ \mathbf{try}\{\ \mathbf{try}\{\mathbf{throw}\}\ \mathbf{catch}\ \{(\kappa, \lambda) : Q_1\}\ \}\ \mathbf{catch}\ \{\kappa : Q_2\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \mid \kappa \hookrightarrow_0 \overline{\kappa} : L$$

Process R either throws an exception in the inner try-catch block (by (THR)) or receives a remote exception (by (RTHR)). By applying (THR), (CLEAN) and (WTHR) in the first case or by (RTHR) in the second case, we have (omitting some queues):

$$\begin{array}{l} R \longrightarrow\equiv\ \mathbf{try}\{\ (\kappa, \lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L\}\ \mathbf{catch}\ \{\kappa : Q_2\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \longrightarrow \\ (\kappa, \lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \longrightarrow (\kappa, \lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \end{array}$$

3.3 Session Types with Interactional Exceptions

In this subsection, we show how to extend the standard type discipline for sessions with interactional exceptions. In comparison with the standard session types, the central difference is the shape of a type itself, which now consists of the abstraction of the default behaviour (the "try" part) and that of the handler behaviour (the "catch" part). This simple extension, combined with the use of levels, allows to establish basic typing properties, guaranteeing that messages are always delivered at proper levels at proper timings in the presence of nested asynchronous escapes, testifying consistency of the operational semantics introduced above.

Type Syntax. The grammar of types extends the standard session types (new parts highlighted with a box $\boxed{\ }$):

$$\begin{array}{rcl} \alpha, \beta & ::= & \downarrow(\theta).\ \alpha \mid \uparrow(\theta).\ \alpha \mid \oplus\{l_i : \alpha_i\}_{i\in I} \mid \&\{l_i : \alpha_i\}_{i\in I} \mid \boxed{\alpha[\![\beta]\!]} \mid \mathtt{end} \mid \mu\mathbf{t}.\ \alpha \mid \mathbf{t} \\ \theta & ::= & \boxed{\langle \alpha[\![\beta]\!]\rangle} \mid \mathtt{bool} \mid \ldots \end{array}$$

α and θ are respectively called *session types* and *service types*. The new type $\alpha\{\!|\beta|\!\}$ (called *try-catch type*) is an abstraction of a try-catch block: in $\alpha\{\!|\beta|\!\}$, α denotes the type of the try-block and β the catch block. A session type α is *plain* if it does not use a try-catch type (except in a service type it carries). We stipulate α and β are both plain in $\alpha\{\!|\beta|\!\}$. This is to prevent a try-catch on κ to occur nested in a catch-block of λ if $\kappa = \lambda$.

The *dual* of a type α, written $\overline{\alpha}$, inverts inputs and outputs [8]. The dual of the try-catch type is defined as $\overline{\alpha\{\!|\beta|\!\}} = \overline{\alpha}\{\!|\overline{\beta}|\!\}$: the other cases are standard. For example, by exchanging input and output, the dual of $\downarrow$ (`string`).`end`$\{\!|\uparrow$ (`bool`).`end`$|\!\}$ is $\uparrow$ (`string`).`end`$\{\!|\downarrow$ (`bool`).`end`$|\!\}$.

Environments. *Typing judgements* for processes and expressions have the forms $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash e : \theta$ respectively where Γ is a *service typing*, which typically maps service (public) channels to service types and Δ is a *session typing* which typically maps session channels to session types. For ($n \in \{0, 1\}$ and $\rho \in \{p, u\}$), typings are defined as

$$\begin{array}{ll} \textit{(Session Typing)} & \Delta ::= \emptyset \mid \Delta,\ \kappa : \alpha \mid \Delta,\ (\kappa, \overline{\kappa}) : \alpha \mid \Delta,\ (\kappa, \overline{\kappa}) : \bot \\ \textit{(Service Typing)} & \Gamma ::= \emptyset \mid \Gamma,\ c : \langle \alpha\{\!|\beta|\!\} \rangle \mid c : \texttt{bool} \mid \Gamma,\ X : \Delta \end{array}$$

In session typings, $\kappa : \alpha$ says that: *at a polarised session channel κ, there is a session of type α*. In the service typing, c either has type $\alpha\{\!|\beta|\!\}$ (a service using a session channel with default behaviour of type α and with a handler of type β) or an atomic type such as `bool`. Typing $X : \Delta$ is used for recursion as in [4].

Typing System for Programs. We show the typing system by which the programmer can check whether her program is error free or not, especially w.r.t. its exception usage. A complete list of typing rules is reported in Table 4.

(TReq) types a request on service channel c whose type, according to Γ, is $\alpha_j\{\!|\beta_j|\!\}$. Condition $s^+ = \kappa_j$ makes sure that the fresh name s^+ will also be in the try-catch after reduction. Session s^+ has type $\overline{\alpha}_j\{\!|\overline{\beta}_j|\!\}$, the dual of c's type. This rule checks that each κ_i in Q (exception handler) has type $\overline{\beta}_i$ (note β_i must be plain) whereas in P it has type $\overline{\alpha}_i\{\!|\overline{\beta}_i|\!\}$ where each $\overline{\beta}_i$ may come from a refinement of κ_i in P. Finally, Γ' is a subset of Γ without free variables for service channels (otherwise the queue may store open terms at run-time). In (TServ), because of SCP, services should never be prefixed therefore the only visible (free) session in P and Q should be s^-.

For (TOut), in $\uparrow (\theta)\mathbf{.}\ \alpha$, the prefixing of a type is read as $(\uparrow (\theta)\mathbf{.}\ \alpha')\{\!|\beta|\!\}$ whenever α has the form $\alpha'\{\!|\beta|\!\}$. Throwing an exception interrupts any conversation, thus (TThr) allows to type **throw** with any $\kappa : \alpha$. (TInact) allows to start from $\mathsf{end}\{\!|\beta|\!\}$ if we are typing in a try-block, while we may want to start from end in a catch-block.

(TPar) requires the coherence relation $\asymp$. Formally, we say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if and only if $\mathsf{fsc}(\Delta_1) \cap \mathsf{fsc}(\Delta_2) = \emptyset$.

Example 6 (Typing Asynchronous and Nested Escapes) The processes in Examples 2 is typable: channel chSeller in both examples has type $\mu\mathbf{t}.\ \uparrow$ (`int`)$\mathbf{.}\ \mathbf{t}\{\!|\downarrow$ (`int`)$\mathbf{.}\ \uparrow$ (`time`)$|\!\}$.

In Example 3, channel chBroker has type $(\downarrow$ (`int`)$\mathbf{.}\ \mu\mathbf{t}.\ \uparrow$ (`int`)$\mathbf{.}\ \mathbf{t})\{\!|\oplus\{\mathtt{l}_1 : \downarrow$ (`int`)$\mathbf{.}\ \uparrow$ (`time`)$, \mathtt{l}_2 : \alpha\}|\!\}$ for some α.

$$\text{(Name)}\ \Gamma, a : \langle\alpha\rangle \vdash a : \langle\alpha\rangle \qquad \text{(Bool)}\ \Gamma \vdash \mathsf{tt}, \mathsf{ff} : \mathsf{bool} \qquad \text{(Or)}\ \frac{\Gamma \vdash e_i : \mathsf{bool}}{\Gamma \vdash e_1 \ \mathsf{or}\ e_2 : \mathsf{bool}}$$

$$\text{(TReq)}\ \frac{\begin{array}{l}\Gamma \vdash P \triangleright \prod_i \kappa_i : \overline{\alpha}_i[\![\overline{\beta}_i]\!] \\ \Gamma' \vdash Q \triangleright \prod_i \kappa_i : \overline{\beta}_i \qquad s^+ = \kappa_j \\ \Gamma \vdash c : \langle\alpha_j[\![\beta_j]\!]\rangle \qquad \Gamma' \subseteq \Gamma,\ \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash \overline{c}(s^+)[\tilde{\kappa}, P, Q] \triangleright \prod_{i\neq j} \kappa_i : \overline{\alpha}_i[\![\overline{\beta}_i]\!]} \qquad \text{(TServ)}\ \frac{\begin{array}{l}\Gamma \vdash P \triangleright s^- : \alpha[\![\beta]\!] \\ \Gamma \vdash Q \triangleright s^- : \beta \qquad \mathsf{fv}(\Gamma) = \emptyset\end{array}}{\Gamma,\ a : \langle\alpha[\![\beta]\!]\rangle \vdash *a(s^-)[P, Q] \rhd \emptyset}$$

$$\text{(TThr)}\ \frac{\mathsf{fv}(\Gamma) = \emptyset}{\Gamma \vdash \mathbf{throw} \triangleright \prod_i \kappa_i : \alpha_i} \qquad \text{(TPar)}\ \frac{\Gamma \vdash P_i \triangleright \Delta_i\ (i = 1, 2)\ \ \Delta_1 \asymp \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \cup \Delta_2}$$

$$\text{(TOut)}\ \frac{\Gamma \vdash e : \theta \qquad \Gamma \vdash P \triangleright \Delta \cdot \kappa : \alpha}{\Gamma \vdash \kappa!\langle e\rangle.\ P \triangleright \Delta \cdot \kappa : \uparrow(\theta).\ \alpha} \qquad \text{(TIn)}\ \frac{\Gamma, x : \theta \vdash P \triangleright \Delta \cdot \kappa : \alpha}{\Gamma \vdash \kappa?(x).\ P \triangleright \Delta \cdot \kappa : \downarrow(\theta).\ \alpha}$$

$$\text{(TSel)}\ \frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa : \alpha_j}{\Gamma \vdash \kappa \triangleleft l_j.\ P \triangleright \Delta \cdot \kappa : \oplus\{l_i : \alpha_i\}_{i\in I}} \qquad \text{(TRes)}\ \frac{\Gamma, a : \langle\alpha[\![\beta]\!]\rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)\, P \triangleright \Delta}$$

$$\text{(TBra)}\ \frac{\Gamma \vdash P_i \triangleright \Delta \cdot \kappa : \alpha_i \quad \forall i \in I}{\Gamma \vdash \kappa \rhd \{l_i : P_i\}_{i\in I} \triangleright \Delta \cdot \kappa : \&\{l_i : \alpha_i\}_{i\in I}} \qquad \text{(TIf)}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathsf{bool} \\ \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta\end{array}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ \ Q \triangleright \Delta}$$

$$\text{(TInact)}\ \frac{\mathsf{fv}(\Gamma) = \emptyset \quad \alpha_i \in \{\mathsf{end}, \mathsf{end}[\![\beta_i]\!]\}}{\Gamma \vdash \mathbf{0} \triangleright \prod_i \kappa_i : \alpha_i} \qquad \text{(TRec)}\ \frac{\Gamma, X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.\ P \triangleright \Delta} \qquad \text{(TVar)}\ \frac{}{\Gamma, X : \Delta \vdash X \triangleright \Delta}$$

Table 4. Typing System for Programs

On Run-Time Processes and Subject Reduction. The ultimate goal of the typing system is to show that errors do not occur (type and communication safety). These results are based on subject reduction i.e. typable process remain typable after reductions. However, the typing system introduced above only provides rules for checking (static) programs. In order to have subject reduction, we also need to provide typing rules for run-time processes namely try-catch blocks, wraps, session restriction and queues. Given the purpose of this lecture note, we shall not address the technical details on how to type run-time processes but take an informal approach instead. We redirect the eager reader to [5].

The basic idea for typing try-catch blocks (and wraps) is to assign a try-catch type $\alpha[\![\beta]\!]$ to channels such that α abstracts the channel usage in the try-block and β the usage in the catch-block. The typing of session restriction is standard. However, the treatment of queues is a little peculiar and the following example will give an intuitive explanation. The process

$$\mathbf{try}\{\,\kappa!\langle\text{"}Hi\text{"}\rangle.\ P\,\}\ \mathbf{catch}\ \{\,\kappa : \kappa!\langle 5\rangle\,\} \mid \kappa \hookrightarrow \overline{\kappa} : \epsilon \tag{1}$$

is such that $\kappa :\uparrow (\texttt{string})[\![\uparrow (\texttt{int})]\!]$. Now, after a reduction step to process

$$\mathbf{try}\{\,P\,\}\ \mathbf{catch}\ \{\,\kappa : \kappa!\langle 5\rangle\,\} \mid \kappa \hookrightarrow \overline{\kappa} : \text{"}Hi\text{"} \tag{2}$$

does the type of κ change? Processes (1) and (2) are identical, except that an output prefix in (1) changes its place to the queue. Thus we can go back from (2) to (1) by placing "*Hi*" on the top of the process. A key idea is to carry out this rollback of a message in typing, using a local type with a hole (a type context) for typing a queue. For example, we type the queue in (2) as the type context $\uparrow\,(\texttt{string})[-]$ where $[-]$ indicates a hole. Now, we can cover the type $\texttt{end}\{\!|\uparrow\,(\texttt{int})|\!\}$ with such a type context, obtaining the original type $\uparrow\,(\texttt{string})\{\!|\uparrow\,(\texttt{int})|\!\}$.

In general, we need to be extra careful when dealing with exception propagation. Queues may contain † which can be preceded and/or followed by other messages. In such cases, all messages sent before the throwing of the exception can be ignored. As an example, if P throws and then outputs 5, (2) reduces to (for some P'):

$$P' \mid \kappa\{\!|\mathbf{0}|\!\} \mid \kappa \hookrightarrow \overline{\kappa} : (5 :: \dagger :: \text{"}Hi\text{"})$$

Above, process $\kappa\{\!|\mathbf{0}|\!\}$ can be typed with the try-catch type $\uparrow\,(\texttt{string})\{\!|\texttt{end}|\!\}$ (in general the try part can be guessed) and, therefore, the composition with the queue type $\uparrow\,(\texttt{int})[-]$ will finally yield $\uparrow\,(\texttt{string})\{\!|\uparrow\,(\texttt{int})|\!\}$. Note that, when applying the queue type to $\uparrow\,(\texttt{string})\{\!|\texttt{end}|\!\}$, we must know that it has to cover the catch part: this can be obviously told by the † in the queue.

In the following Theorem, $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Theorem 3 (Subject Reduction). *Let P be a program such that $\Gamma \vdash P \triangleright \emptyset$. If $P \longrightarrow^* Q$ then $\Gamma \vdash Q \triangleright \emptyset$.*

As a corollary, the typing system also satisfies type safety and communication safety including communication-error freedom and linearity [10, Theorem 5.5].

4 Multiparty Asynchronous Session Types

4.1 Preview on Multiparty Interactions

In general, session types do not allow to abstract from inter-session causality which could be useful at a designing stage. As an example, let us consider a simple refinement of the Buyer-Seller protocol [4]: consider two buyers, Buyer1 and Buyer2, wish to buy an expensive product, say a book, from Seller by combining their money:

1. Buyer1 sends the title of the book to Seller;
2. Seller sends to both Buyer1 and Buyer2 its quote;
3. Buyer1 tells Buyer2 how much she can pay, and Buyer2 either *accepts* or *rejects* the quote by notifying Seller.

It is extremely awkward (if logically possible) to decompose this scenario into three binary sessions, between Buyer1 and Seller, between Buyer2 and Seller, and between Buyer1 and Buyer2. Abstracting this protocol as three separate session types also means that our type abstraction loses essential sequencing information in this interaction scenario: for instance, we may want to guarantee that Buyer2 accepts only after Seller has

sent a quote to Buyer1. For validating this conversation scenario as a whole, therefore, the conversation structure should be represented as a *single session*.

Many existing business protocols including financial protocols are written as a collaboration of several peers. Typical message-passing parallel algorithms also frequently demand distribution of a request to, and collection of the results from, many peers. All these usecases are most naturally abstracted as a single session. In this section, we adress a generalisation of the foregoing binary session types to multiparty asynchronous sessions [10, 1, 2, 14].

4.2 The π-Calculus with Multiparty Asynchronous Session Types

Syntax. In the sequel, we shall keep the same notation as the one introduced for interactional exceptions. Let e be the set of expression defined as in Section 3. Then *processes* (programs) are given by the following grammar:

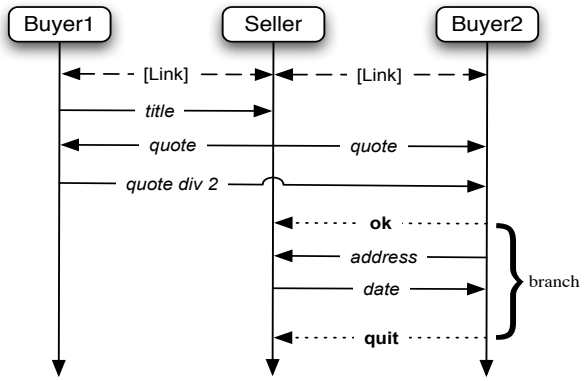
P	::=	$a_{[\mathtt{p}]}(\tilde{s})\mathbf{.}\ P$	(accept)	$\mid\ \overline{a}_{[2..\mathtt{n}]}(\tilde{s})\mathbf{.}\ P$	(request)
	$\mid$	$s?(\tilde{x})\mathbf{.}\ P$	(input)	$\mid\ s!\langle\tilde{e}\rangle\mathbf{.}\ P$	(output)
	$\mid$	$s?(\!(\tilde{s})\!)\mathbf{.}\ P$	(session reception)	$\mid\ s!\langle\!\langle\tilde{s}\rangle\!\rangle\mathbf{.}\ P$	(delegation)
	$\mid$	$s \rhd \{l_i : P_i\}_{i\in I}$	(branch)	$\mid\ s \lhd l\mathbf{.}\ P$	(select)
	$\mid$	$P \mid Q$	(par)	$\mid\ \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q$	(cond)
	$\mid$	$\mathbf{0}$	(inact)	$\mid\ (\nu a)\ P$	(resServ)
	$\mid$	X	(termVar)	$\mid\ \mu X\mathbf{.}\ P$	(recursion)

Most of the primitives above are identical to the ones we saw for interactional exceptions except from (accept), (request), (delegation) and (session reception). The prefix $\overline{a}_{[2..\mathtt{n}]}(\tilde{s})\mathbf{.}\ P$ initiates a new session through a, by distributing a vector of freshly generated session channels $\tilde{s}$ to the remaining $\mathtt{n}-1$ participants, each of shape $a_{[\mathtt{p}]}(\tilde{s})\mathbf{.}\ Q_{\mathtt{p}}$ for $2 \leq \mathtt{p} \leq \mathtt{n}$. All receive $\tilde{s}$, over which the actual session communications can now take place among the $\mathtt{n}$ parties. $\mathtt{p}$, $\mathtt{q}$,... range over natural numbers called *participants* of a session.

Session communications are performed using the primitives we saw for interactional exceptions but also allowing for session delegation. In delegation, the capability to participate in a session is delegated to the session receiver by passing the whole channels associated with the session. Note that session communication is polyadic (apart from branch/select).

The notions of bound and free identifiers, channels, alpha equivalence $\equiv_\alpha$ and substitution are standardly adapted to the calculus with multuparty sessions.

Example 7 (Two Buyer Protocol) The Two Buyer protocol can be represented by the following diagram:



Above, Buyer1 sends a book title to Seller, then Seller sends back a quote to Buyer1/2; Buyer1 now tells Buyer2 how much she can contribute, and Buyer2 notifies Seller if it accepts the quote or not. We now describe the behaviour of Buyer1 as a process:

$$\mathsf{Buyer1} \stackrel{\text{def}}{=} \overline{a}_{[2,3]}(b_1, b_2, b_2', s_1, s_2).\quad s_1!\langle\text{“War and Peace”}\rangle.$$
$$b_1?(\mathit{quote}).\quad b_2'!\langle \mathit{quote}\ \mathsf{div}\ 2\rangle.\ P_1$$

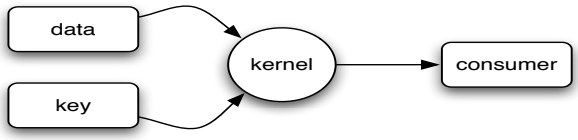
Channel b_1 is for Buyer1 to receive messages: b_2 and b_2' for Buyer2 and s_1 and s_2 for Seller. Buyer1 above is willing to contribute to half of the quote. In P_1, Buyer1 may perform the remaining transactions with Seller and Buyer2. The remaining participants follow.

$$\begin{aligned}\mathsf{Buyer2} \stackrel{\text{def}}{=}\ & a_{[2]}(b_1, b_2, b_2', s_1, s_2).\quad b_2?(\mathit{quote}).\quad b_2'?(\mathit{contrib}).\\ & \textbf{if}\ (\mathit{quote} - \mathit{contrib} \le 99)\\ & \quad \textbf{then}\quad s_2 \lhd \mathsf{ok}.\ s_2!\langle \mathit{address}\rangle; b_2?(x).\ P_2\\ & \quad \textbf{else}\quad s_2 \lhd \mathsf{quit}.\ \mathbf{0}\end{aligned}$$

$$\begin{aligned}\mathsf{Seller} \stackrel{\text{def}}{=}\ & a_{[3]}(b_1, b_2, b_2', s_1, s_2).\quad s_1?(\mathit{title}).\quad b_1, b_2!\langle \mathit{quote}\rangle.\\ & s_2 \rhd \{\mathsf{ok}\colon\ s_2?(x).\ b_2!\langle \mathit{date}\rangle; Q\,,\quad \mathsf{quit}\colon\ \mathbf{0}\}\end{aligned}$$

Above $b_1, b_2!\langle v\rangle.\ P$ stands for $b_1!\langle v\rangle.\ .\ b_2!\langle v\rangle.\ P$, assuming b_1, b_2 are distinct: due to asynchrony there is in effect no order among the sending actions at b_1, b_2. Note that Buyer2 needs to use two input channels, b_2 and b_2' while, for Seller, s_1 and s_2 are not necessary. The first input (for *quote*) is from Seller, while the second one (for *contrib*) is from Buyer1. Hence there is no guarantee that they arrive in a fixed order, as can be easily seen by analysing reduction paths (this is Lamport's principle [13]). Thus if we were to use b_2 for both actions, the two messages can be confused, losing linear usage of a channel. Later we shall show our type discipline can avoid such an error.

Example 8 (A Streaming Protocol) We next consider a simple protocol for the standard stream cipher [15].



Data Producer and Key Producer continuously send a data stream and a key stream respectively to Kernel. Kernel calculates their XOR and sends the result to Consumer.

Assuming streams are sent block by block (say as large arrays), we can realise this protocol as communicating processes. We only focus on communication behaviour. The kernel initiates a session:

$$\mathsf{Kernel} \stackrel{\text{def}}{=} \overline{a}_{[2,3,4]}(d,k,c).\ \mu X.\ d?(x).\ k?(y).\ c!\langle x \ \mathsf{xor}\ y\rangle.\ X$$

The channels d and k are used for Kernel to receive data and keys from Data Producer and Key Producer, respectively, while c is used for Consumer to receive the encrypted data from Kernel. Data Producer and Consumer can be given as:

$$\mathsf{DataProducer} \stackrel{\text{def}}{=} a_{[2]}(d,k,c).\ \mu Y.\ d!\langle data\rangle.\ Y \qquad \mathsf{Consumer} \stackrel{\text{def}}{=} a_{[3]}(d,k,c).\ \mu Z.\ c?(data).\ Z$$

Key Producer is identical to Data Producer except it outputs at k instead of d. When three processes are composed, we can verify that, although processes repeatedly send and receive data using the same channels, messages are always consumed in the order they are produced, an essential requirement for correctness of the protocol. This is because each channel is used by exactly one sender. We shall show how this argument can be cleanly represented and validated through session types.

Reduction Semantics. Because of asynchrony, similarly to the case of exceptions, we need to introduce a run-time syntax[4]:

$$P ::= \ \ldots \mid (\nu s)\ P \qquad \text{(resSess)} \qquad \mid s : L \qquad \text{(queue)}$$

$$L \ ::= \ \epsilon \mid h :: L \qquad h \ ::= \ l \mid \tilde{v} \mid \tilde{s} \qquad v \ ::= \ a \mid \mathsf{tt} \mid \mathsf{ff}$$

The run-time syntax is almost identical to the interactional exceptions case exept from the queues. In this case, queues are not polarised (we have no polarised channels) and one message in transit can carry a tuple of values or session names. In the sequel, the term $(\nu s_1)\ \ldots (\nu s_k)\ P$ may be denoted by $(\nu \tilde{s})\ P$.

The semantics has a standard *structural congruence* relation which is defined as the smallest congruence relation on processes such that $(P, \mid)$ is a commutative monoid and includes the standard rules for restriction (such as scope extrusion) and recursion.

The reduction semantics is given by the *reduction relation*, defined as the smallest relation on processes generated by the rules in Table 5. (Init) describes a session initiation among $\mathtt{n}$-parties through synchronisation, generating m fresh session channels and the associated m empty queues. As a result $\mathtt{n}$ participants now share the newly generated m channels, hence their queues. Note that, in general, the number of threads ($\mathtt{n}$) can be different from that of session channels (m), giving flexibility in channel usage.

Similarly to Section 3, (Out), (Deleg) and (Sel) respectively enqueue values, channels and a label in the queue for s. (In), (SRec) and (Bra) dequeue values, channels and a label. (Bra) further selects the corresponding branch. Other rules are standard.

[4] Note we do not need contexts because there are no nesting operators

(Init)	$\overline{a}_{[2..n]}(\tilde{s})\mathbf{.}\ P_1 \mid a_{[2]}(\tilde{s})\mathbf{.}\ P_2 \mid \cdots \mid a_{[n]}(\tilde{s})\mathbf{.}\ P_n \ \rightarrow\ (\nu\tilde{s})\,(P_1 \mid P_2 \mid ... \mid P_n \mid s_1 : \epsilon \mid \cdots \mid s_m : \epsilon)$
(Out)	$s!\langle\tilde{e}\rangle\mathbf{.}\ P \mid s : L \rightarrow P \mid s : (\tilde{v} :: L) \qquad (\tilde{e} \downarrow \tilde{v})$
(Deleg)	$s!\langle\!\langle\tilde{t}\rangle\!\rangle\mathbf{.}\ P \mid s : L \rightarrow P \mid s : (\tilde{t} :: L)$
(Sel)	$s \lhd l\mathbf{.}\ P \mid s : L \rightarrow P \mid s : (l :: L)$
(In)	$s?(\tilde{x})\mathbf{.}\ P \mid s : (L :: \tilde{v}) \ \rightarrow\ P[\tilde{v}/\tilde{x}] \mid s : L$
(SRec)	$s?(\!(\tilde{t})\!)\mathbf{.}\ P \mid s : (L :: \tilde{t}) \ \rightarrow\ P \mid s : L$
(Bra)	$s \rhd \{l_i : P_i\}_{i\in I} \mid s : (L :: l_j) \ \rightarrow\ P_j \mid s : L \qquad (j \in I)$
(If)	$\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \ \rightarrow\ P \qquad (e \downarrow \mathsf{tt}) \qquad \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \ \rightarrow\ Q \qquad (e \downarrow \mathsf{ff})$
(Res)	$P \rightarrow P' \quad \wedge \quad n \in \{a, s\} \ \Rightarrow\ (\nu n)\,P \rightarrow (\nu n)\,P'$
(Par)	$P \rightarrow P' \ \Rightarrow\ P \mid Q \ \rightarrow\ P' \mid Q$
(Str)	$P \equiv P'$ and $P' \rightarrow Q'$ and $Q' \equiv Q \ \Rightarrow\ P \rightarrow Q$

Table 5. Reduction

4.3 Types for Multiparty Sessions

Developing programs for multiparty sessions demands a clear formal design as to how multiple participants communicate and synchronise with each other. To program individual participants without such a design and hope they somehow realise a meaningful and error-free conversation is hardly practical, especially for team programming. In binary session types the type for an endpoint also served as the description of the whole conversation, but this is no longer possible for multiparty sessions. This is why we need the type abstraction which describes global conversation scenarios of multiparty sessions. This is achieved by defining global types, a multiparty session abstraction based on the notion of *choreography* [4]: a type no longer describe the usage of a channel from a one participant viewpoint, but will give a vantage perspective of the whole session (and its session channels). An intuitive example is given by the diagram in the two buyer protocol which gives us a global view, or choreography, of how the session would run. However, having a global description may be useful at designing stage, but needs a correspondent description of the local behaviour of each participant in the session. The process of generating each participant local behaviour (or local session type) is called end-point projection (EPP) or, simply, *projection* [4]. For instance, the EPP of Buyer1's behaviour from the diagram in the two buyer protocol would be a sequence of an output (book title), an input (quote) and, finally, an output (quote halved).

Once we are capable of projecting global descriptions into end-point behaviour, we can consider the following development steps for programs with multiparty sessions:

1. A programmer describes an intended interaction scenario as a global type G;
2. she develops code, one for each participant, incrementally validating its conformance to the projection of G onto each participant by efficient type-checking.

When programs are executed, their interactions are guaranteed to follow the stipulated scenario. The type specification also serves as a basis for maintenance and upgrade.

Global Types. A *global session type*, or *global type*, denoted $G, G', \ldots$, is given by the following grammar:

$$\begin{array}{rcll} G & ::= & \mathtt{p} \to \mathtt{p}' : k\,\langle\theta\rangle.G' & \text{(values)} \\ & | & \mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J} & \text{(branching)} \\ & | & G, G' & \text{(parallel)} \\ & | & \mu\mathbf{t}.G & \text{(recursive)} \\ & | & \mathbf{t} & \text{(variable)} \\ & | & \mathsf{end} & \text{(end)} \end{array}$$

$$\theta \ ::= \ \tilde{S} \ \mid \ \alpha@\mathtt{p} \qquad S \ ::= \ \mathsf{bool} \ \mid \ \mathsf{nat} \ \mid \ \ldots \ \mid \ \langle G \rangle$$

Type $\mathtt{p} \to \mathtt{p}' : k\,\langle\theta\rangle.G'$ says that participant $\mathtt{p}$ sends a message of type θ to channel k (represented as a finite natural number) received by participant $\mathtt{p}'$ and interactions described in G' take place. We assume that in each prefix from $\mathtt{p}$ to $\mathtt{p}'$ we have $\mathtt{p} \neq \mathtt{p}'$, i.e. we prohibit reflexive interaction. θ ranges over *value types* $\tilde{S}$ or *local types* α paired with participant names. Each value type is a vector of types for shared names called *sorts*. A local type α, whose details will be addressed in the next subsection, may hereby used for delegation of session channels. Type $\mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J}$ says participant $\mathtt{p}$ sends one of the labels to channel k which is then received by participant $\mathtt{p}'$. If l_j is sent, interactions described in G_j take place.

Type G, G' represents concurrent run of interactions specified by G and G'. Type $\mu\mathbf{t}.G$ is a recursive type for recurring conversation structures, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e. type variables only appear under the prefixes. As in standard session types, we take an *equi-recursive* view, not distinguishing between $\mu\mathbf{t}.G$ and its unfolding $G[\mu\mathbf{t}.G/\mathbf{t}]$. We assume that $\langle G \rangle$ in the grammar of sorts is closed, i.e. without type variables. Type end represents the termination of the session. We identify "G, end" and "end, G" with G.

We stipulate that, in a global type G, each channel can only be used, one or more times, among two fixed parties, one party using it for input/session reception/branching while the other party for output/delegation/selection. This condition is not restrictive and dispenses with the need for linearity check to ensure well-formedness of global types found in [10] (see [2] for details).

Example 9 (A Global Type for the Two Buyer Protocol) We write principals and channels with legible symbols though they are actually numbers: $\mathtt{Bi} = i$, $\mathtt{S} = 3$, $b_1 = 1$, $b_2 = 2$, $b_2' = 3$, $s_1 = 4$ and $s_2 = 5$. The following is a global type for the two buyer protocol:

$$\mathtt{B1} \to \mathtt{S} : s_1\langle\mathsf{string}\rangle. \quad \mathtt{S} \to \mathtt{B1} : b_1\langle\mathsf{int}\rangle. \quad \mathtt{S} \to \mathtt{B2} : b_2\langle\mathsf{int}\rangle. \quad \mathtt{B1} \to \mathtt{B2} : b_2'\langle\mathsf{int}\rangle.$$
$$\mathtt{B2} \to \mathtt{S} : s_2 \left\{ \begin{array}{l} \mathsf{ok} : \mathtt{B2} \to \mathtt{S} : s_2\,\langle\mathsf{string}\rangle.\mathtt{S} \to \mathtt{B2} : b_2\,\langle\mathsf{date}\rangle.\mathsf{end}, \\ \mathsf{quit} : \mathsf{end} \} \end{array} \right.$$

The type gives a vantage view of the whole conversation scenario.

Example 10 (A Global Type for the Streaming Protocol) In this example, we present the global type of the simple streaming protocol. Below we unfold its recursion once, and set: $d = 1$, $k = 2$, $c = 3$, $\mathtt{K} = 1$, $\mathtt{DP} = 2$, $\mathtt{C} = 3$ and $\mathtt{KP} = 4$.

$$\begin{array}{l} \mu\mathbf{t}. \ \ \mathtt{DP} \to \mathtt{K} : d\,\langle\mathsf{bool}\rangle. \ \ \mathtt{KP} \to \mathtt{K} : k\,\langle\mathsf{bool}\rangle. \ \ \mathtt{K} \to \mathtt{C} : c\,\langle\mathsf{bool}\rangle. \\ \mathtt{DP} \to \mathtt{K} : d\,\langle\mathsf{bool}\rangle. \ \ \mathtt{KP} \to \mathtt{K} : k\,\langle\mathsf{bool}\rangle. \ \ \mathtt{K} \to \mathtt{C} : c\,\langle\mathsf{bool}\rangle.\mathbf{t} \end{array}$$

Local Types. *Local session types* or *local types*, ranged over by $\alpha, \beta, ..$, are types for local behaviour of processes in a multiparty session, acting as a link between global types and processes (they have many analogies to standard session types) and are defined by the following grammar:

$$\alpha \ ::= \quad k!\langle\theta\rangle.\,\alpha \ \mid \ k?\langle\theta\rangle.\,\alpha \ \mid \ k \oplus \{l_i\colon \alpha_i\}_{i\in I} \ \mid \ k \,\&\, \{l_i\colon \alpha_i\}_{i\in I} \ \mid \ \mu\mathbf{t}.\,\alpha \ \mid \ \mathbf{t} \ \mid \ \mathsf{end}$$

All constructs come from standard binary session types except from the following major changes for multiparty interactions:

- Since a session now uses multiple channels, a session type needs to record the identity (number) of a session channel it uses at each action type as found in [4].
- Since a type is inferred for each participant, we use the notation $\alpha@\mathtt{p}$ (*located type*) representing a local type α assigned to participant $\mathtt{p}$. A located type is also used for delegation.

Type $k?\langle\theta\rangle.\,\alpha$ represents the behaviour of inputting values of type θ while $k!\langle\theta\rangle.\,\alpha$ is for sending. Types $k \,\&\, \{l_i\colon \alpha_i\}_{i\in I}$ and $k \oplus \{l_i\colon \alpha_i\}_{i\in I}$ are respectively for branching and select at k. The rest is the same as the global types, demanding type variables occur guarded by a prefix and taking an equi-recursive approach for recursive types. Note local types α do not contain parallel composition like in [4].

Projection. In the introduction to this section, we have discussed the need for a projection of global types into local behaviour. The following is the formal definition of such a projection:

Definition 4 (Projection). Let G be linear. Then the *projection of G onto* $\mathtt{p}$, written $G \upharpoonright \mathtt{p}$, is inductively given as:

$$(\mathtt{p}_1 \to \mathtt{p}_2\colon k\,\langle\theta\rangle.G') \upharpoonright \mathtt{p} = \begin{cases} k!\langle\theta\rangle.(G' \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_1 \neq \mathtt{p}_2 \\ k?\langle\theta\rangle.(G' \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_2 \neq \mathtt{p}_1 \\ (G' \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} \neq \mathtt{p}_2 \wedge \mathtt{p} \neq \mathtt{p}_1 \end{cases}$$

$$(\mathtt{p}_1 \to \mathtt{p}_2\colon k\,\{l_j\colon G_j\}_{j\in J}) \upharpoonright \mathtt{p} = \begin{cases} k \oplus \{l_j\colon (G_j \upharpoonright \mathtt{p})\}_{j\in J} & \text{if } \mathtt{p} = \mathtt{p}_1 \neq \mathtt{p}_2 \\ k\&\{l_j\colon (G_j \upharpoonright \mathtt{p})\}_{j\in J} & \text{if } \mathtt{p} = \mathtt{p}_2 \neq \mathtt{p}_1 \\ (\bigsqcup_{i\in I} G_i \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} \neq \mathtt{p}_2 \wedge \mathtt{p} \neq \mathtt{p}_1 \\ & \text{and } \forall i, j \in I. G_i \upharpoonright \mathtt{p} \bowtie G_j \upharpoonright \mathtt{p} \end{cases}$$

$$(G_1, G_2) \upharpoonright \mathtt{p} = \begin{cases} G_i \upharpoonright \mathtt{p} & \text{if } \mathtt{p} \in G_i \text{ and } \mathtt{p} \notin G_j,\ i \neq j \in \{1,2\} \\ \mathsf{end} & \text{if } \mathtt{p} \notin G_1 \text{ and } \mathtt{p} \notin G_2 \end{cases}$$

$$(\mu\mathbf{t}.G) \upharpoonright \mathtt{p} = \mu\mathbf{t}.(G \upharpoonright \mathtt{p}) \qquad \mathbf{t} \upharpoonright \mathtt{p} = \mathbf{t} \qquad \mathsf{end} \upharpoonright \mathtt{p} = \mathsf{end}$$

Whenever the projection is defined, G is said to be *projectable*.

The mapping is intuitive. We regard the map to act on the syntax of global types. In parallel composition, $\mathtt{p}$ should be contained in at most a single type, ensuring each type is single-threaded. In the branching, all projections should generate an identical local type (otherwise undefined) up to mergeability $\bowtie$. Mergeability [4], not present in the original work on multiparty session types [10], is the smallest equivalence over local types closed under all type contexts and the rule:

$$\frac{\forall i \in (I \cap J).\ \alpha_i \bowtie \beta_i \qquad \forall i \in I \backslash J.\ \forall j \in J \backslash I.\ l_i \neq l_j}{k \,\&\, \{l_i \colon \alpha_i\}_{i \in I} \bowtie k \,\&\, \{l_j \colon \beta_j\}_{j \in J}}$$

Intuitively, the mergeability condition requires two local types to be identical except from branches &, where branches with different labels may be different.

The projection of branching is then defined as the merging $\sqcup$ of the projections of the branches. Formally, $\alpha \sqcup \beta$ is a partial commutative operator over local types which is well-defined iff $\alpha \bowtie \beta$ and is an isomorphism except from the following case:

$$k \,\&\, \{l_i \colon \alpha_i\}_{i \in I} \sqcup k \,\&\, \{l_j \colon \beta_j\}_{j \in J} \;=\; k\&(\{l_i \colon \alpha_i \sqcup \beta_i\}_{i \in I \cap J} \cup \{l_i \colon \alpha_i\}_{i \in I} \cup \{l_j \colon \beta_j\}_{j \in J})$$

Using the merging operator above allows for more global types to have a projection. In fact, we can also write global types where, for instance, in a binary branching from $\mathtt{p}_1$ to $\mathtt{p}_2$, a third participant $\mathtt{p}_3$ can behave differently depending on the selection made by $\mathtt{p}_1$. This is *only* allowed when p_3 can be projected with branching local type: each branch corresponds to a branch in the global type and may be selected by p_2 (or some other causally notified participant) according to p_1's selection. The following is an example of projection and clarifies the usefulness of merging.

Example 11 Consider the following global type:

$$\begin{array}{l} \mathtt{p}_1 \to \mathtt{p}_2 : k\,\{ \\ \quad \mathsf{ok} : \mathtt{p}_2 \to \mathtt{p}_3 : k'\,\{\mathsf{paymore} : \ldots\}, \\ \quad \mathsf{quit} : \mathtt{p}_2 \to \mathtt{p}_3 : k'\,\{\mathsf{refund} : \ldots\} \\ \} \end{array}$$

Above, $\mathtt{p}_1$ selects, over k, ok or quit. Based on this selection $\mathtt{p}_2$ will either select paymore or refund. The projection of $\mathtt{p}_1$ is the local type $k \oplus \{\mathsf{ok} \colon \ldots, \mathsf{quit} \colon \ldots,\}$ while $\mathtt{p}_2$ is projected as:

$$k\&\{\quad \mathsf{ok} \colon k' \oplus \{\mathsf{paymore} \colon \ldots\}, \quad \mathsf{quit} \colon k' \oplus \{\mathsf{reject} \colon \ldots\} \quad \}$$

However, the projection of $\mathtt{p}_3$ on the ok branch is $k'\&\{\mathsf{paymore} \colon \ldots\}$ whereas, on the quit branch, it is $k'\&\{\mathsf{refund} \colon \ldots\}$. Such a projection is not allowed in [10], however, we can easily merge the two local types, yielding:

$$k'\&\{\ \mathsf{paymore} \colon \ldots,\ \mathsf{refund} \colon \ldots\ \}$$

Example 12 The following global type is *not* projectable:

$$\mathtt{A} \to \mathtt{B} : k\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{bool} \rangle,\ \mathsf{quit} : \mathtt{C} \to \mathtt{D} : k'\langle \mathsf{nat} \rangle\}$$

Intuitively, when we project this type onto C or D, regardless of the choice made by A, they should behave in the same way: participants C and D should be independent threads. If we change the above nat to bool as: $\mathtt{A} \to \mathtt{B} : k\{\mathsf{ok} : \mathtt{C} \to \mathtt{D} : k'\langle\mathsf{bool}\rangle, \mathsf{quit} : \mathtt{C} \to \mathtt{D} : k'\langle\mathsf{bool}\rangle\}$, we can define the coherent projection as follows:

$$\{\, k \oplus \{\mathsf{ok} : \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}@\mathtt{A},\ k\&\{\mathsf{ok} : \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}@\mathtt{B},\ k'!\langle\mathsf{bool}\rangle@\mathtt{C},\ k'?\langle\mathsf{bool}\rangle@\mathtt{D}\,\}$$

Environments. Assuming global types are projectable, judgements are shaped like the ones for interactional exception:

$$\begin{array}{rcl} \Gamma & ::= & \emptyset \mid \Gamma, u : S \mid \Gamma, X : \Delta \\ \Delta & ::= & \emptyset \mid \Delta \cdot \tilde{s} : \{\alpha@\mathtt{p}\}_{\mathtt{p}\in I} \end{array}$$

The *service environment* (also called *sorting*) Γ is a finite map from names to sorts and from process variables to session environment. A *session environment* Δ records linear usage of session channels. In the binary sessions, it assigned a type to a single channel; now it assigns a family of located types to a vector of session channels. We write $\tilde{s} : \alpha@\mathtt{p}$ for a singleton typing $\tilde{s} : \{\alpha@\mathtt{p}\}$.

Therefore, judgements have the shape $\Gamma \vdash P \triangleright \Delta$ which reads: "under the environment Γ, process P has typing Δ".

Typing System for Programs. The type system for programs is given in Table 6. Note that if we set $|\tilde{s}| = 1$ and $\mathtt{n} = 2$, and delete p from located type, the shape of rules is essentially identical with the original binary session typing [19].

We shall now comment the rules that differ from standard binary session typing.

In (TReq), the rule for the session request, the type for $\tilde{s}$ is the *first* projection of the declared global type for a in Γ. Similarly, when typing session accept with (TAcc), we take the p-th projection. The local type $(G \upharpoonright \mathtt{p})@\mathtt{p}$ means that the participant p has $G \upharpoonright \mathtt{p}$, which is the projection of G onto p, as its local type. The condition $|\tilde{s}| = \max(\mathsf{sid}(G))$ ensures the number of session channels meets those in G. The typing $\tilde{s} : \alpha@\mathtt{p}$ (which stands for $\tilde{s} : \{\alpha@\mathtt{p}\}$) ensures each prefix does not contain parallel threads sharing $\tilde{s}$. Both rules, (TReq) and (TAcc), are applicable whenever G is projectable.

(TOut) and (TIn) are the rules for sending and receiving values. Since the k-th name s_k of $\tilde{s}$ is used as the subject, we record the number k. In both rules, "p" in $\alpha@\mathtt{p}$ ensures that P is (being inferred as) the behaviour for participant p, and its domain should be $\tilde{s}$. Then the relevant type prefixes ($k!\langle\tilde{S}\rangle$ for the output and $k?\langle\tilde{S}\rangle$ for the input) are composed in the conclusion's session environment.

(TDeleg) and (TSRec) are the rules for delegation of a session and its dual (not present in Section 3). Delegation of a multiparty session passes the whole capability to participate in a multiparty session: thus operationally we send the whole vector of session channels. The carried type α' is located, making sure that the behaviour by the receiver at the passed channels takes the role of a specific participant (here $\mathtt{p}'$) in the delegated multiparty session. The rest follows the standard delegation rule [19], observing (TDeleg) says that $\tilde{t} : \alpha'@\mathtt{p}'$ does not appear in P symmetrically to (TSRec) which uses the channels in P.

(TSel) is the rule for selection, and identical with the one used for interactional exceptions. In (TBranch), the type may have less branches than the actual ones occurring

$$\text{(Name)}\ \Gamma, a : S \vdash a : S \qquad \text{(Bool)}\ \Gamma \vdash \mathsf{true}, \mathsf{false} : \mathsf{bool} \qquad \text{(Or)}\ \frac{\Gamma \vdash e_i \triangleright \mathsf{bool}}{\Gamma \vdash e_1 \mathsf{or}\ e_2 : \mathsf{bool}}$$

$$\text{(TReq)}\ \frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright 1)@1 \quad |\tilde{s}| = \max(\mathsf{sid}(G))}{\Gamma \vdash \overline{a}_{[2..n]}(\tilde{s}).\ P \triangleright \Delta}$$

$$\text{(TAcc)}\ \frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright \mathtt{p})@\mathtt{p} \quad |\tilde{s}| = \max(\mathsf{sid}(G))}{\Gamma \vdash a_{[\mathtt{p}]}(\tilde{s}).\ P \triangleright \Delta}$$

$$\text{(TOut)}\ \frac{\forall j.\ \Gamma \vdash e_j : S_j \qquad \Gamma \vdash\ P \triangleright \Delta \cdot \tilde{s} : \alpha@\mathtt{p}}{\Gamma \vdash\ s_k!\langle \tilde{e} \rangle.\ P \triangleright \Delta \cdot \tilde{s} : k!\langle \tilde{S} \rangle.\ \alpha@\mathtt{p}} \qquad \text{(TIn)}\ \frac{\Gamma, x : \tilde{S} \vdash\ P \triangleright \Delta \cdot \tilde{s} : \alpha@\mathtt{p}}{\Gamma \vdash\ s_k?(\tilde{x}).\ P \triangleright \Delta \cdot \tilde{s} : k?\langle \tilde{S} \rangle.\ \alpha@\mathtt{p}}$$

$$\text{(TDeleg)}\ \frac{\Gamma \vdash\ P \triangleright \Delta \cdot \tilde{s} : \alpha@\mathtt{p}}{\Gamma \vdash\ s_k!\langle\!\langle \tilde{t} \rangle\!\rangle.\ P \triangleright \Delta \cdot \tilde{s} : k!\langle \alpha'@\mathtt{p}' \rangle.\ \alpha@\mathtt{p} \cdot\ \tilde{t} : \alpha'@\mathtt{p}'}$$

$$\text{(TSRec)}\ \frac{\Gamma \vdash\ P \triangleright \Delta \cdot \tilde{s} : \alpha@\mathtt{p} \cdot \tilde{t} : \alpha'@\mathtt{p}'}{\Gamma \vdash\ s_k\ ?(\!(\tilde{t})\!).\ P \triangleright \Delta \cdot \tilde{s} : k?\langle \alpha'@\mathtt{p}' \rangle.\ \alpha@\mathtt{p}}$$

$$\text{(TSel)}\ \frac{\Gamma \vdash\ P \triangleright \Delta \cdot \tilde{s} : \alpha_j@\mathtt{p} \qquad j \in I}{\Gamma \vdash\ s_k \lhd l_j.\ P \triangleright \Delta \cdot \tilde{s} : k \oplus \{l_i : \alpha_i\}_{i \in I}@\mathtt{p}} \qquad \text{(TBra)}\ \frac{\Gamma \vdash\ P_j \triangleright \Delta \cdot \tilde{s} : \alpha_j@\mathtt{p} \qquad \forall j \in J \qquad I \subseteq J}{\Gamma \vdash\ s_k \rhd \{l_j : P_j\}_{i \in J} \triangleright \Delta \cdot \tilde{s} : k\ \&\ \{l_i : \alpha_i\}_{i \in I}@\mathtt{p}}$$

$$\text{(TPar)}\ \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta' \qquad \Delta \sim \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \qquad \text{(TIf)}\ \frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \triangleright \Delta}$$

$$\text{(TInact)}\ \frac{\Delta\ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \text{(TRes)}\ \frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)\ P \triangleright \Delta} \qquad \text{(TVar)}\ \frac{}{\Gamma, X : \Delta \vdash X \triangleright \Delta} \qquad \text{(TRec)}\ \frac{\Gamma, X : \Delta \vdash\ P \triangleright \Delta}{\Gamma \vdash\ \mu X.\ P \triangleright \Delta}$$

Table 6. Typing System for Expressions and Processes

in the process: this still ensures that a selection is never made on a branch that does not exist [4]. This change in the rule was made to allow merging. The original work on multiparty session types [10] adopts a rule like the one we used for exceptions.

(TPar) uses $\asymp$ to ensure well-formedness of the session typing, taking a the disjoint union of each local type. The partial operator $\circ$ is defined as:

$$\{\alpha_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I} \circ \{\alpha'_{\mathtt{p}'}@\mathtt{p}'\}_{\mathtt{p}' \in J} = \{\alpha_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I} \cup \{\alpha'_{\mathtt{p}'}@\mathtt{p}'\}_{\mathtt{p}' \in J}$$

if $I \cap J = \emptyset$. Then we say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \sim \Delta_2$, if for all $\tilde{s}_i \in \mathsf{dom}(\Delta_i)$ such that $\tilde{s}_1 \cap \tilde{s}_2 \neq \emptyset$, $\tilde{s} = \tilde{s}_1 = \tilde{s}_2$ and $\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s})$ is defined. When $\Delta_1 \sim \Delta_2$, the *composition of* Δ_1 *and* Δ_2, written $\Delta_1 \circ \Delta_2$, is given as:

$$\begin{aligned} \Delta_1 \circ \Delta_2 = \{&\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s}) \mid \tilde{s} \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \\ &\cup \Delta_1 \setminus \mathsf{dom}(\Delta_2) \cup \Delta_2 \setminus \mathsf{dom}(\Delta_1) \end{aligned}$$

In (TInact) and (TVar), "end only" means Δ only contains end as session types.

Example 13 (Two Buyer Protocol) In the two buyer protocol, write Buyer1 as process $\overline{a}_{[2,3]}(b_1, b_2, b_2', s_1, s_2)\mathbf{.}\ Q_1$ and Buyer2 as $a_{[2]}(b_1, b_2, b_2', s_1, s_2)\mathbf{.}\ Q_2$. Then Q_1 and Q_2 have the following typing under $\Gamma = \{a : \langle G \rangle\}$ where G is given in the corresponding example in pag. 18, letting $\mathtt{Bi} = i$, $\mathtt{S} = 3$, $b_1 = 1$, $b_2 = 2$, $b_2' = 3$, $s_1 = 4$ and $s_2 = 5$ and assuming P_1, P_2, Q are $\mathbf{0}$:

$$\Gamma \vdash Q_1 \triangleright \tilde{s} : s_1!\langle \mathsf{string} \rangle\mathbf{.}\ b_1?\langle \mathsf{int} \rangle\mathbf{.}\ b_2'!\langle \mathsf{int} \rangle@\mathtt{B1}$$
$$\Gamma \vdash Q_2 \triangleright \tilde{s} : b_2?\langle \mathsf{int} \rangle\mathbf{.}\ b_2'?\langle \mathsf{int} \rangle\mathbf{.}\ s_2 \oplus \{\mathsf{ok} : s_2!\langle \mathsf{string} \rangle\mathbf{.}\ b_2?\langle \mathsf{date} \rangle\mathbf{.}\ \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}@\mathtt{B2}$$

Similarly for Seller. After prefixing at a, we can compose all three by (TPar).

Example 14 (The Streaming Protocol) We let $\Gamma = \{a : \langle G' \rangle\}$ where G' is given in Example 9. Let $d = 1$, $k = 2$, $c = 3$, $\mathtt{K} = 1$, $\mathtt{DP} = 2$, $\mathtt{C} = 3$ and $\mathtt{KP} = 4$. Write R_1, R_2, R_3 and R_4 for the processes which are under the initial prefix (at the shared name) of Kernel, DataProducer, Consumer and KeyProducer, respectively. Then we can type each agent as:

$$\Gamma \vdash R_1 \triangleright dkc : \mu\mathbf{t}.d?\langle \mathsf{bool} \rangle; k?\langle \mathsf{bool} \rangle; c!\langle \mathsf{bool} \rangle; \mathbf{t}@\mathtt{K}$$
$$\Gamma \vdash R_2 \triangleright dkc : \mu\mathbf{t}.d!\langle \mathsf{bool} \rangle; \mathbf{t}@\mathtt{DP} \qquad \Gamma \vdash R_4 \triangleright dkc : \mu\mathbf{t}.c?\langle \mathsf{bool} \rangle; \mathbf{t}@\mathtt{C}$$

(R_4 is similar as R_2). Note these types correspond to the projection of G' onto respective participants: thus Kernel, DataProducer, Consumer and KeyProducer are typable programs, which can be composed to make the initial configuration.

Example 15 (An Example of Delegation) One source of the expressiveness of the session types comes from a facility of *delegation* (often called *higher-order session passing*). We will type and see the relationship with global and local types. Consider the following three participants:

$$\mathsf{Alice} \stackrel{\text{def}}{=} \overline{a}_{[2]}(t_1, t_2)\mathbf{.}\ \overline{b}_{[2,3]}(s_1, s_2)\mathbf{.}\ t_1!\langle\!\langle s_1, s_2 \rangle\!\rangle\mathbf{.}\ \mathbf{0}$$
$$\mathsf{Bob} \stackrel{\text{def}}{=} a_{[2]}(t_1, t_2)\mathbf{.}\ b_{[1]}(s_1, s_2)\mathbf{.}\ t_1?(\!(s_1, s_2)\!)\mathbf{.}\ s_1!\langle 1 \rangle; \mathbf{0}$$
$$\mathsf{Carol} \stackrel{\text{def}}{=} b_{[2]}(s_1, s_2)\mathbf{.}\ s_1?(x); P$$

where Alice delegates its capability to Bob. Since there are two multicasting, there are two global specifications, one for a and another for b as follows:

$$G_a = \mathtt{A} \to \mathtt{B} : t_1\ \langle s_1!\langle \mathsf{int} \rangle@\mathtt{A} \rangle.\mathsf{end}$$
$$G_b = \mathtt{A} \to \mathtt{C} : s_1\ \langle \mathsf{int} \rangle.\mathsf{end}$$

where the type $s_1!\langle \mathsf{int} \rangle@\mathtt{A}$ means the capability to send an integer from participant A via channel s_1. This capability is passed to B so that B behaves as A. However, since two specifications are independent, C does not have to know who would pass the capability.

Example 16 (Addition Protocol [1]) In this protocol, a client Client sends two natural numbers n and m to a server Addition and waits for a reply containing the sum $n + m$. Addition reacts to Client's messages as follows: if the second operand is 0 then it sends the first operand n back to Client as a result, otherwise it sends n and m to a third

participant called SuccPred which will reply with $n + 1$ and $m - 1$ and then send a looping message to Client. This behaviour is repeated until the second operand becomes 0. Starting from the global type G, we have:

$$\texttt{Client} \to \texttt{Addition}\colon k_1\,\langle int\rangle.\quad \texttt{Client} \to \texttt{Addition}\colon k_1\,\langle int\rangle.$$

$$\mu\mathbf{t}.\ \texttt{Addition} \to \texttt{SuccPred} : k_2$$
$$\left\{\begin{array}{l} \mathsf{tt} : \texttt{Addition} \to \texttt{Client}\colon k_3\,\{\mathsf{ok}\colon \texttt{Addition} \to \texttt{Client}\colon k_3\,\langle int\rangle.\mathsf{end}\} \\ \mathsf{ff} : \texttt{Addition} \to \texttt{SuccPred}\colon k_2\,\langle int, int\rangle.\texttt{SuccPred} \to \texttt{Addition}\colon k_4\,\langle int, int\rangle. \\ \quad\ \texttt{Addition} \to \texttt{Client}\colon k_3\,\{\mathsf{wait}\colon \mathsf{t}\} \end{array}\right\}$$

The projection of G generates the following local types:

$$G \upharpoonright \texttt{Client} \;=\; k_1!\langle int\rangle.\ k_1!\langle int\rangle.\ \mu t.\ k_3\&\{\mathsf{ok} : k_3?\langle int\rangle.\ \mathsf{end},\ \mathsf{wait} : \mathsf{t}\}$$

$$G \upharpoonright \texttt{Addition} \;=\; k_1?\langle int\rangle.\ k_1?\langle int\rangle.\ \mu t.\ k_2 \oplus \left\{\begin{array}{l} \mathsf{tt} : k_3 \oplus \{\mathsf{ok} : k_3!\langle int\rangle.\mathsf{end}\}, \\ \mathsf{ff} : k_2!\langle int, int\rangle.k_4?\langle int, int\rangle.k_3 \oplus \{\mathsf{wait} : \mathsf{t}\} \end{array}\right\}$$

$$G \upharpoonright \texttt{SuccPred} \;=\; \mu t.\ k_2\&\{\mathsf{tt} : \mathsf{end},\ \mathsf{ff} : k_2?\langle int, int\rangle.k_4!\langle int, int\rangle.\mathsf{t}\}$$

and, finally, the protocol can be implemented as:

$$\begin{array}{lcl} \mathsf{Client} & = & \overline{a}_{[2,3]}(k_1, k_2, k_3, k_4).\ k_1!\langle n\rangle.\ k_1!\langle m\rangle.\ \mu X.\ k_3 \triangleright \{\mathsf{ok} : k_3?(x),\ \mathsf{wait} : X\} \\ \mathsf{Addition} & = & a_{[2]}(k_1, k_2, k_3, k_4).\ k_1?(x_1).\ k_1?(x_2). \\ & & \quad \mu X.\ \textbf{if}\ x_2 = 0 \\ & & \qquad \textbf{then}\ k_2 \triangleleft \mathsf{tt}.\ k_3 \triangleleft \mathsf{ok}.\ k_3!\langle x_1\rangle \\ & & \qquad \textbf{else}\ k_2 \triangleleft \mathsf{ff}.\ k_2!\langle x_1, x_2\rangle.\ k_4?(x_1, x_2).\ k_3 \triangleleft \mathsf{wait}.\ X \\ \mathsf{SuccPred} & = & a_{[3]}(k_1, k_2, k_3, k_4).\ \mu X.\ k_2 \triangleright \{\mathsf{tt} : \mathbf{0},\ \mathsf{ff} : k_2?(x, y).\ k_4!\langle x + 1, y - 1\rangle\} \end{array}$$

where a is the shared name for the protocol, Client = 1, Addition = 2 and SuccPred = 3. Note that we had to introduce a synchronisation between Addition and Client at each iteration. This avoids the case when Client waits forever in case of negative m hence violating safety (in such case the global type would not be projectable).

On Run-Time Processes and Subject Reduction. Similarly to what we have discussed for asynchronous exceptions, also multiparty session types enjoy a subject reduction property.

Informally, we need to extend the typing rules to include those for runtime processes which involve message queues in a fashion similar to what discussed for asynchronous interactional exceptions. .

Theorem 5 (Subject Reduction). *$\Gamma \vdash P \triangleright \emptyset$ and $P \to P'$ imply $\Gamma \vdash P' \triangleright \emptyset$.*

By the correspondence between local types and global types, these results guarantee that interactions between typed processes exactly follow the conversation scenario specified in a global type. Also in this case, *safety* and *session fidelity* follow. Also, under a certain condition we can also have *progress* [10].

5 Discussion

5.1 Interactional Exceptions

Comparing to the original work in [5], we have omitted the typing of run-time processes which, although important, is only used as a technique for proving type safety. The original work also addresses the problem of termination for try-catch blocks. For instance, suppose there are two (and only two) processes in a configuration, which are try-catch blocks and which are communicating in a session. If each party's default process becomes the inaction process, it is natural to reduce each try-catch block to the inaction, freeing up the resources for its handler. This garbage collection is essential when we consider integration of interactional exceptions into the standard imperative programming languages with sequential composition since in this case launching a handler depends on whether a process reduces to the inaction or not. In [5], it is shown that a well-typed process satisfies a liveness condition with such a garbage collector.

As discussed in [5], programs can be extended such that in the session initialisation processes $*c(\lambda)[P, Q]$ and $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$, the handler Q may contain another try-catch at the same λ (currently, try-catch is only used at run-time). Such an extension of the formalism would allow a process to "try" again after an exception has been thrown (cascading exceptions). For this purpose, try-catch types should be extended such that in $\alpha\{\!|\beta|\!\}$ the type α is always plain while β can be either plain or a try-catch type. With essentially the same operational semantics, this generalised calculus satisfies the subject reduction and liveness properties.

The key idea of the presented operational semantics is the use of exception levels in queues and their interplay with wrapped processes. In implementation, the queue level can be recorded in a header of each message which its receiver can check efficiently. The wrapping level can be a part of a process state, recording its exception depth. Various optimisations are possible, for example dispensing with most coordination protocols when the handler type is trivial, obtaining essentially the same level of efficiency as local exception.

Session delegations are not allowed in [5] but can be formulated by storing frozen processes in queues. The type soundness holds by extending the typing rules with those in [8].

A further generalisation to multiparty session types for flexible multicast exception propagation is currently being investigated.

5.2 Multiparty Session Types

Multiparty session types have been further investigated since the original work in [10]. The theory presented in this lecture note mainly differs from [10] in two points: (i) session channels can only be used between two fixed parties (in one direction), first discovered in [2] and (ii) the introduction of the merging operator for allowing a broader set of projectable global types (first proposed here).

Bejleri and Yoshida [1] have studied multiparty *synchronous* session types, a variant where communication is synchronous (no queues) as in [8]. They introduce multicasting, higher-order communication via multi-polarity labels and an alternative definition

of delegation in global types. The work in [2] develops a type system (built up on the original one) for global progress in multiparty sessions: well typed terms guarantee the absence of deadlock. A very recent work, [14], introduces communication subtyping, which allows for partial commutativity of actions, providing flexibility and safe optimisation. The authors propose an algorithm for the subtyping relation, which can calculate conformance of end-point processes to an agreed global specification. Moreover, they introduce an algorithm for abstracting a global specification from end-point processes allowing programmer to choose between a top-down and a bottom-up style of communication programming.

There are several significant future topics on the theory and applications of multiparty session types. This generalised session type structure is currently being used as one of the formal foundations of the next version of a web service description language, WS-CDL from W3C [17] and a message scheme for financial protocols, UNIFI from ISO [12]. Another topic is the use of this theory as a basis of communication-centred extensions of general purpose programming languages [11]. Others include tools assistance for the design and elaboration of global types; and integration of the type discipline with diverse specification concerns including security and assertional methods.

References

1. A. Bejleri and N. Yoshida. Synchronous multiparty session types. In *In Proceedings of Programming Languages Approaches to Concurrency and Communication-Centric Software (PLACES'08)*, 2008.
2. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, pages 418–433, 2008.
3. E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256. Springer, 2008.
4. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
5. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions for session types. In *19th International Conference on Concurrency Theory (Concur'08)*, LNCS, pages 402–417. Springer, 2008.
6. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
7. S. Gay and V. T. Vasconcelos. Asynchronous functional session types. TR 2007–251, University of Glasgow, May 2007.
8. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
9. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, 91:165–185, 2007.
10. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
11. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, LNCS. Springer, 2008. To appear.
12. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org/index.cfm?item_id=56664#interest`.

13. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
14. D. Mostrous, N. Yoshida, and K. Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions, 2009. To appear in Proc. of ESOP'09.
15. B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1993.
16. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
17. Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.
18. Wikipedia. Exception handling. `http://en.wikipedia.org/wiki/Exception_handling`, 2009.
19. N. Yoshida and V. T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisit. *ENTCS*, 171(4):73–93, 2007.